Dense Arithmetic over Finite Fields with CUMODP

Sardar Anisul Haque\textsuperscript{1}  Xin Li\textsuperscript{2}  Farnam Mansouri\textsuperscript{1}  
Marc Moreno Maza\textsuperscript{1}  Wei Pan\textsuperscript{3}  Ning Xie\textsuperscript{1}

\textsuperscript{1}University of Western Ontario, Canada  
\textsuperscript{2}Universidad Carlos III, Spain  
\textsuperscript{3}Intel Corporation, Canada

ICMS, August 8, 2014
Outline

1 Overview

2 A many-core machine model for minimizing parallelism overheads

3 Putting the MCM model into practice

4 Adaptive algorithms

5 Bivariate system solving

6 Conclusion
Background

Reducing everything to multiplication

- Polynomial multiplication and matrix multiplication are at the core of many algorithms in symbolic computation.
- Algebraic complexity is often estimated in terms of multiplication time.
- At the software level, this reduction is also common (Magma, NTL).
- Can we do the same for SIMD-multithreaded algorithms?

Building blocks in scientific software

- The Basic Linear Algebra Subprograms (BLAS) is an inspiring and successful project providing low-level kernels in linear algebra, used by LINPACK, LAPACK, MATLAB, Mathematica, Julia (among others).
- Other BB’s successful projects: FFTW, SPIRAL (among others).
- The GNU Multiple Precision Arithmetic Library project plays a similar role for rational numbers and floating-point numbers.
- No symbolic computation software dedicated to sequential polynomial arithmetic managed to play the unification role of the BLAS.
- Could this work in the case of GPUs?
CUMODP Mandate (1/2)

Functionalities

- **Level 1**: basic arithmetic operations that are specific to a polynomial representation or a coefficient ring: multi-dimensional FFTs/TFTs, converting integers from CRA to mixed-radix representations

- **Level 2**: basic arithmetic operations for dense or sparse polynomials with coefficients in $\mathbb{Z}/p\mathbb{Z}$, $\mathbb{Z}$ or in floating point numbers: polynomial multiplication, polynomial division

- **Level 3**: advanced arithmetic operations on families of polynomials: operations based on subproduct trees, computations of subresultant chains.
Targeted architectures

- Graphics Processing Units (GPUs) with code written in CUDA,
- CUMODP aims at supporting BPAS (Basic Polynomial Algebra Subprograms) which targets multi-core processors and which is written in CilkPlus.
- Thanks to our Meta_Fork framework, automatic translation between CilkPlus and OpenMP are possible, as well as conversions to C/C++.
- Unifying code for CUMODP and BPAS is conceivable (see the SPIRAL project) but highly complex (multi-core processors enforce memory consistency while GPUs do not, etc.)
<table>
<thead>
<tr>
<th><strong>Algorithm choice</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Level 1</strong> functions (n-D FFTs/TFTs) are highly optimized in terms of arithmetic count, locality and parallelism.</td>
</tr>
<tr>
<td><strong>Level 2</strong> functions provide several algorithms or implementation for the same operation: coarse-grained &amp; fine-grained, plain &amp; FFT-based.</td>
</tr>
<tr>
<td><strong>Level 3</strong> functions combine several Level 2 algorithms for achieving a given task.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Implementation techniques</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>At <strong>Level 2</strong>, the user can choose between algorithms minimizing work or algorithms maximizing parallelism.</td>
</tr>
<tr>
<td>At <strong>Level 3</strong>, this leads to adaptive algorithms that select appropriate Level 2 functions depending on available resources (number of cores, input data size).</td>
</tr>
</tbody>
</table>
Code organization

Main features

- developers and users have access to the same code.
- mainly 32bit arithmetic so far
- Regression tests and benchmark scripts are also distributed.
- Documentation is generated by doxygen.
- A manually written documentation is work in progress.
- Three student theses and about 10 papers present & document CUMODP.
Outline

1. Overview

2. A many-core machine model for minimizing parallelism overheads

3. Putting the MCM model into practice

4. Adaptive algorithms

5. Bivariate system solving

6. Conclusion
Similarly to a CUDA program, an MMM program specifies for each kernel the number of thread-blocks and the number of threads per thread-block.
An MMM program $\mathcal{P}$ is a directed acyclic graph (DAG), called kernel DAG of $\mathcal{P}$, whose vertices are kernels and edges indicate serial dependencies.

Since each kernel of the program $\mathcal{P}$ decomposes into a finite number of thread-blocks, we map $\mathcal{P}$ to a second graph, called thread-block DAG of $\mathcal{P}$, whose vertex set consists of all thread-blocks of $\mathcal{P}$.
Machine parameters

\textbf{Z}: Size (expressed in machine words) of the local memory of any SM.

\textbf{U}: Time (expressed in clock cycles) to transfer one machine word between the global memory and the local memory of any SM.

(Assume \( U > 1 \).)

Thus, if \( \alpha \) and \( \beta \) are the maximum number of words respectively read and written to the global memory by a thread of a thread-block, then the data transferring time \( T_D \) between the global and local memory of a thread-block satisfies \( T_D \leq (\alpha + \beta) U \).
For any kernel $\mathcal{K}$ of an MMM program,

- **work** $W(\mathcal{K})$ is the total number of local operations of all its threads;
- **span** $S(\mathcal{K})$ is the maximum number of local operations of one thread.

For the entire program $\mathcal{P}$,

- **work** $W(\mathcal{P})$ is the total work of all its kernels;
- **span** $S(\mathcal{P})$ is the longest path, counting the weight (span) of each vertex (kernel), in the kernel DAG.
For any kernel $\mathcal{K}$ of an MMM program,

- **parallelism overhead** $O(\mathcal{K})$ is the total data transferring time among all its thread-blocks.

For the entire program $\mathcal{P}$,

- **parallelism overhead** $O(\mathcal{P})$ is the total parallelism overhead of all its kernels.
A many-core machine model: running time estimates

A Graham-Brent theorem with parallelism overhead

**Theorem.** Let $K$ be the maximum number of thread blocks along an anti-chain of the thread-block DAG of $\mathcal{P}$. Then the running time $T_\mathcal{P}$ of the program $\mathcal{P}$ satisfies:

$$T_\mathcal{P} \leq \left( \frac{N(\mathcal{P})}{K} + L(\mathcal{P}) \right) C(\mathcal{P}),$$

where

- $N(\mathcal{P})$: number of vertices in the thread-block DAG,
- $L(\mathcal{P})$: critical path length (where length of a path is the number of edges in that path) in the thread-block DAG.

Note: The Graham-Brent theorem, a greedy scheduler operating with $P$ processors executes a multithreaded computation with work $T_1$ and span $T_\infty$ in time

$$T_P \leq T_1/P + T_\infty.$$
Outline

1. Overview

2. A many-core machine model for minimizing parallelism overheads

3. Putting the MCM model into practice

4. Adaptive algorithms

5. Bivariate system solving

6. Conclusion
Applying the MCM model

Applying MCM to minimize parallelism overheads by determining an appropriate value range for a given program parameter

- In each case, a program \( P(s) \) depends on a parameter \( s \) which varies in a range \( S \) around an initial value \( s_0 \);
- the work ratio \( W_{s_0}/W_s \) remains essentially constant, meanwhile the parallelism overhead \( O_s \) varies more substantially, say \( O_{s_0}/O_s \in \Theta(s - s_0) \);
- Then, we determine a value \( s_{\text{min}} \in S \) maximizing the ratio \( O_{s_0}/O_s \).
- Next, we use our version of Graham-Brent theorem to check whether the upper bound for the running time of \( P(s_{\text{min}}) \) is less than that of \( P(s_o) \). If this holds, we view \( P(s_{\text{min}}) \) as a solution of our problem of algorithm optimization (in terms of parallelism overheads).
Applying MCM to the polynomial division operation

Naive division algorithm of a thread-block with \( s = 1 \): each kernel performs 1 division step

\[
\frac{W_1}{W_s} = \frac{8(Z+1)}{9Z+7}, \quad \frac{O_1}{O_s} = \frac{20}{441}Z, \quad \frac{T_1}{T_s} = \frac{(3+5U)Z}{3(Z+21U)}.
\]

\( T_1/T_s > 1 \) if and only if \( Z > 12.6 \) holds, which clearly holds on actual GPU architectures.

Thus, the optimized algorithm (that is for \( s > 1 \)) is overall better than the naive one (that is for \( s = 1 \)).
MCM applied to plain univariate polynomial multiplication and the Euclidean algorithm (1/2)

Applying MCM to the plain multiplication and the Euclidean algorithm

- For plain polynomial multiplication, this analysis suggested to minimize $s$.

- For the Euclidean algorithm, our analysis suggested to maximize the program parameter $s$.

- Both are verified experimentally.
MCM applied to plain univariate polynomial multiplication and the Euclidean algorithm (1/2)

CUMODP vs NTL

- CUMODP: plain but parallel algorithms
- NTL: asymptotically fast FFT-based but serial algorithms
- Our experimentations are executed on a NVIDIA Tesla M2050 GPU and an Intel Xeon X5650 CPU.

CUMODP plain polynomial division vs NTL FFT-based polynomial division.

CUMODP plain Euclidean algorithm vs NTL FFT-based polynomial GCD.
FFT-based multiplication

Our GPU implementation relies on Stockham FFT algorithm

- Let $d$ be the degree, then $n = 2^\left\lceil \log_2(2^d - 1) \right\rceil$.

- Based on the MCM, the work is $15n \log_2(n) + 2n$, the span is $15n + 2$, and the parallelism overhead is $(36n + 21)U$.

<table>
<thead>
<tr>
<th>Size</th>
<th>CUMODP</th>
<th>FLINT</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>$2^{12}$</td>
<td>0.0032</td>
<td>0.003</td>
<td></td>
</tr>
<tr>
<td>$2^{13}$</td>
<td>0.0023</td>
<td>0.008</td>
<td>3.441</td>
</tr>
<tr>
<td>$2^{14}$</td>
<td>0.0039</td>
<td>0.013</td>
<td>3.346</td>
</tr>
<tr>
<td>$2^{15}$</td>
<td>0.0032</td>
<td>0.023</td>
<td>7.216</td>
</tr>
<tr>
<td>$2^{16}$</td>
<td>0.0065</td>
<td>0.045</td>
<td>6.942</td>
</tr>
<tr>
<td>$2^{17}$</td>
<td>0.0084</td>
<td>0.088</td>
<td>10.475</td>
</tr>
<tr>
<td>$2^{18}$</td>
<td>0.0122</td>
<td>0.227</td>
<td>18.468</td>
</tr>
<tr>
<td>$2^{19}$</td>
<td>0.0198</td>
<td>0.471</td>
<td>23.738</td>
</tr>
<tr>
<td>$2^{20}$</td>
<td>0.0266</td>
<td>1.011</td>
<td>27.581</td>
</tr>
<tr>
<td>$2^{21}$</td>
<td>0.0718</td>
<td>2.086</td>
<td>29.037</td>
</tr>
<tr>
<td>$2^{22}$</td>
<td>0.1451</td>
<td>4.419</td>
<td>30.454</td>
</tr>
<tr>
<td>$2^{23}$</td>
<td>0.3043</td>
<td>9.043</td>
<td>29.717</td>
</tr>
</tbody>
</table>

**Table**: Running time (in sec.) for FFT-based polynomial multiplication: CUMODP vs FLINT.
Outline

1. Overview
2. A many-core machine model for minimizing parallelism overheads
3. Putting the MCM model into practice
4. Adaptive algorithms
5. Bivariate system solving
6. Conclusion
Parallelizing subproduct-tree techniques

Challenging in parallel implementation:

- The divide-and-conquer formulation of operations is not sufficient to provide enough parallelism.
- One must parallelize the underlying polynomial arithmetic operations.
- The degrees of the involved polynomials vary greatly during the course of the execution of operations (subproduct tree, evaluation or interpolation).
- So does the work load of the tasks, which makes those algorithms complex to implement on many-core GPUs.
Subproduct trees

Subproduct tree technique

- Split the point set $U = \{u_0, \ldots, u_{n-1}\}$ with $n = 2^k$ into two halves of equal cardinality and proceed recursively with each of the two halves.

- This leads to a complete binary tree $M_n$ of depth $k$ having the points $u_0, \ldots, u_{n-1}$ as leaves.

- Let $m_i = x - u_i$ and define each non-leaf node in the binary tree as a polynomial multiplication

  $$M_{i,j} = m_j \cdot 2^i \cdot m_j \cdot 2^{i+1} \cdots m_j \cdot 2^{i+2^i-1} = \prod_{0 \leq \ell < 2^i} m_j \cdot 2^i + \ell.$$  

- For each level of the binary tree smaller than a threshold $H$, we compute the subproducts using plain multiplication.

- For each level of the binary tree larger than the threshold $H$, we compute the subproducts using FFT-based multiplication.
Subinverse tree

- Associated with the subproduct tree $M_n \ (n = 2^k)$, the subinverse tree $InvM_n$ is a complete binary tree with the same height as $M_n$.

- For $j$-th node of level $i$ in $InvM_n$ for $0 \leq i \leq k$, $0 \leq j < 2^{k-i}$,

$$InvM_{i,j} \cdot \text{rev}_{2^i}(M_{i,j}) \equiv 1 \mod x^{2^i},$$

where $\text{rev}_{2^i}(M_{i,j}) = x^{2^i} M_{i,j}(1/x)$. 

Multi-point evaluation & interpolation: estimates

Given a univariate polynomial $f \in K[x]$ of degree less than $n = 2^k$ and evaluation points $u_0, \ldots, u_{n-1} \in K$, compute $(f(u_0), \ldots, f(u_{n-1}))$. (Recall the threshold $H$ in the subproduct (subinverse) tree.)

- the work is $O(n \log_2^2(n) + n \log_2(n) + n 2^H)$,
- the span is $O(\log_2^2(n) + \log_2(n) + 2^H)$, and
- the parallelism overhead is $O((\log_2^2(n) + \log_2(n) + H) U)$.

Given distinct points $u_0, \ldots, u_{n-1} \in K$ and arbitrary values $v_0, \ldots, v_{n-1} \in K$, compute the unique polynomial $f \in K[x]$ of degree less than $n = 2^k$ that takes the value $v_i$ at the point $u_i$ for all $i$.

- the work is $O(n \log_2^2(n) + n \log_2(n) + n 2^H)$,
- the span is $O(\log_2^2(n) + \log_2(n) + 2^H)$, and
- the parallelism overhead is $O(\log_2^2(n) + \log_2(n) + H)$. 
## Multi-point evaluation & interpolation: benchmarks

<table>
<thead>
<tr>
<th>Deg.</th>
<th>CUMODP</th>
<th>FLINT</th>
<th>SpeedUp</th>
<th>CUMODP</th>
<th>FLINT</th>
<th>SpeedUp</th>
</tr>
</thead>
<tbody>
<tr>
<td>$2^{14}$</td>
<td>0.2034</td>
<td>0.17</td>
<td></td>
<td>0.2548</td>
<td>0.22</td>
<td></td>
</tr>
<tr>
<td>$2^{15}$</td>
<td>0.2415</td>
<td>0.41</td>
<td>1.6971</td>
<td>0.3073</td>
<td>0.53</td>
<td>1.7242</td>
</tr>
<tr>
<td>$2^{16}$</td>
<td>0.3126</td>
<td>0.99</td>
<td>3.1666</td>
<td>0.4026</td>
<td>1.26</td>
<td>3.1294</td>
</tr>
<tr>
<td>$2^{17}$</td>
<td>0.4285</td>
<td>2.33</td>
<td>5.4375</td>
<td>0.5677</td>
<td>2.94</td>
<td>5.1780</td>
</tr>
<tr>
<td>$2^{18}$</td>
<td>0.7106</td>
<td>5.43</td>
<td>7.6404</td>
<td>0.9034</td>
<td>6.81</td>
<td>7.5379</td>
</tr>
<tr>
<td>$2^{19}$</td>
<td>1.0936</td>
<td>12.63</td>
<td>11.5484</td>
<td>1.3931</td>
<td>15.85</td>
<td>11.3768</td>
</tr>
<tr>
<td>$2^{20}$</td>
<td>1.9412</td>
<td>29.2</td>
<td>15.0420</td>
<td>2.4363</td>
<td>36.61</td>
<td>15.0268</td>
</tr>
<tr>
<td>$2^{21}$</td>
<td>3.6927</td>
<td>67.18</td>
<td>18.1923</td>
<td>4.5965</td>
<td>83.98</td>
<td>18.2702</td>
</tr>
<tr>
<td>$2^{22}$</td>
<td>7.4855</td>
<td>153.07</td>
<td>20.4486</td>
<td>9.2940</td>
<td>191.32</td>
<td>20.5851</td>
</tr>
<tr>
<td>$2^{23}$</td>
<td>15.796</td>
<td>346.44</td>
<td>21.9321</td>
<td>19.6923</td>
<td>432.13</td>
<td>21.9441</td>
</tr>
</tbody>
</table>

**Table:** Running time (in sec.) on NVIDIA Tesla C2050 for multi-point evaluation and interpolation: CUMODP vs FLINT.
GPU support for bivariate system solving

Bivariate polynomial system solver (based on the theory of *regular chains*)

- Polynomial subresultant chains are calculated in CUDA.
- Univariate polynomial GCDs are computed in C either by means of the plain Euclidean algorithm or an asymptotically fast algorithm.

<table>
<thead>
<tr>
<th>System</th>
<th>Pure C</th>
<th>Mostly CUDA code</th>
<th>SpeedUp</th>
</tr>
</thead>
<tbody>
<tr>
<td>dense-70</td>
<td>5.22</td>
<td>0.50</td>
<td>10.26</td>
</tr>
<tr>
<td>dense-80</td>
<td>6.63</td>
<td>0.77</td>
<td>8.59</td>
</tr>
<tr>
<td>dense-90</td>
<td>8.39</td>
<td>1.16</td>
<td>7.19</td>
</tr>
<tr>
<td>dense-100</td>
<td>19.53</td>
<td>1.80</td>
<td>10.79</td>
</tr>
<tr>
<td>dense-110</td>
<td>21.41</td>
<td>2.57</td>
<td>8.33</td>
</tr>
<tr>
<td>dense-120</td>
<td>25.71</td>
<td>3.48</td>
<td>7.39</td>
</tr>
<tr>
<td>sparse-70</td>
<td>0.89</td>
<td>0.31</td>
<td>2.81</td>
</tr>
<tr>
<td>sparse-80</td>
<td>3.64</td>
<td>1.18</td>
<td>3.09</td>
</tr>
<tr>
<td>sparse-90</td>
<td>3.13</td>
<td>0.92</td>
<td>3.40</td>
</tr>
<tr>
<td>sparse-100</td>
<td>8.86</td>
<td>1.20</td>
<td>7.38</td>
</tr>
</tbody>
</table>

Table: Running time (in sec.) for bivariate system solving over a small prime field
Outline

1. Overview
2. A many-core machine model for minimizing parallelism overheads
3. Putting the MCM model into practice
4. Adaptive algorithms
5. Bivariate system solving
6. Conclusion
Summary

www.cumodp.org